

A general-purpose modifiable AST parser

Michael Doberstein

Bachelorarbeit

Beginn der Arbeit:	29.09.2024
Abgabe der Arbeit:	30.12.2024
Gutachter:	Dr. John Witulski Dr. Jens Bendisposto

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30.12.2024

Michael Doberstein

Abstract

Software developers are often facing the issue of maintaining large amount of source code. To keep the source code up to date, often times a process called refactoring is needed, in which the software developers restructure source code. Refactorings do not change the behaviour of a program.

To allow the software developers to perform refactorings automatically, most integrated development environments (IDE) provide tools to perform predefined refactorings automatically. The effort of performing some refactorings on large code bases can be too much to perform manually. Most of the time, the tools provided by the IDE are sufficient to perform the refactorings, but sometimes a custom solution is required. One way to create automated custom refactorings is by parsing the source code and building it into an abstract syntax tree (AST), which can be manipulated.

For this thesis a Java library was created, which can parse a subset of the context-free languages based on their corresponding grammar definitions and generates an AST based on a grammar definition. The AST provides several methods to search and modify the tree. The library can convert the AST back to source code at the end of modifications, while keeping most of the formatting and comments of the original source file.

The program provides a relatively easy method of safe large scale refactorings, however the effort of defining a grammar is of considerable size. It has been shown, that this tool can also be used to extend a given programming language and allow for additional features to be added, basically allowing for a primitive form of source to source compilation.

Erklärung zur Nutzung generativer KI

Im Rahmen der vorliegenden Bachelorarbeit wurde generative KI zu folgenden Zwecken genutzt:

- DeepL Write zur sprachlichen Überarbeitung der Arbeit
- Machine learning assistant von der IntelliJ IDE wurde bei Implementierung des Programmes verwendet

Acknowledgements

First and foremost, I would like to express my gratitude to my family, especially my mother, Gabriele Doberstein, and my brothers, Alexander and Christian Doberstein. Your support and encouragement throughout my studies have meant a great deal to me and have helped me stay focused during this journey.

I would also like to thank my thesis advisor, Dr. John Witulski, for his guidance and support during the development of this project. The weekly meetings and your feedback were invaluable in helping me write this thesis and the developing the software.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Related work	3
2	Theoretical background	3
2.1	Regular Expressions and Grammars	3
2.2	Lexer	5
2.3	Parser	6
3	Implementation	9
3.1	Architecture	9
3.2	Class generation vs general classes	11
3.3	Grammar file syntax	11
3.4	CST-Generation	13
3.5	AST-Generation	14
3.5.1	list-Modifier	15
3.5.2	alias-Modifier	16
3.5.3	hidden-Modifier	16
3.5.4	inline-Modifier	17
3.5.5	Not implemented modifiers	17
3.6	AST search methods	18
3.7	Selectors	18
3.8	AST Modifications	20
3.9	Code generation	20
3.10	Extendability	21
3.10.1	Lexer	21

3.10.2 AST Generation	21
3.10.3 Selectors	22
4 Implemented Grammars	23
4.1 MiniJava	23
4.2 Extended MiniJava	23
5 Use cases	24
5.1 Yoda conditions	24
5.2 Translation system	25
5.3 Code transpilation	26
5.4 Code analysis	27
6 Conclusion and further work	28
List of Figures	29
List of Tables	29
References	30

1 Introduction

For this thesis a Java library called `gp-modifiable-ast` was implemented. `gp` stands for general purpose, meaning that it can provide the API for different grammars, e.g. for different programming languages.

1.1 Motivation

The motivation for this work came from a practical problem. During the development of a large enterprise web application a change of the translation system was needed. A translation system receives a token as a string and returns a predefined translation depending on the desired language. This caused some problems because the new translation system had a different call syntax than the old one. This refactoring required looking at external data stored in a database for each different call to the translation function. Because of the dependency on external data, the automatic tools provided by the IDE were not sufficient to refactor these calls.

This would have been a very time-consuming task to do manually, as there were over ten thousand calls to this function. Therefore another solution was needed.

To solve this problem, a special JavaScript parser like Esprima [esp] was used. Esprima can parse JavaScript, provide an AST, allow modifications on the AST, and convert back to source code. With this library, it was possible to perform the refactoring with the necessary customization options and accuracy.

However, tools like Esprima exist mostly for popular programming languages, and each tool provides a different API.

While there are parser generators like SableCC [sab] that can generate parsers from a definition file, they are intended to be used for a different purpose. Common parser generators are implemented to provide part of a compiler front end, but they do not need to maintain whitespace, comments, and be able to generate the original source code from the AST.

The goal of this work is to provide a library called `gp-modifiable-ast` to minimize the effort needed to be able to perform manipulations directly on the AST and to be able to transform the AST back into source code while maintaining all whitespaces and comments.

The following example is intended to illustrate that rewritable abstract syntax trees can be used for refactorings that require some knowledge of the program structure. The change that should be made to the following source code is to rename all variables to enforce a camel case naming scheme for variables, while keeping snake case for function names. While this can be done with IDE tools, it would require manual effort each time.

Implementing a custom solution can enforce this naming scheme automatically.

```

1 void should_be_snake_case(int should_be_camel_case) {
2     int second_variable = should_be_camel_case;
3 }

```

Listing 1: Example of untransformed code

The resulting AST from this code might look like this:

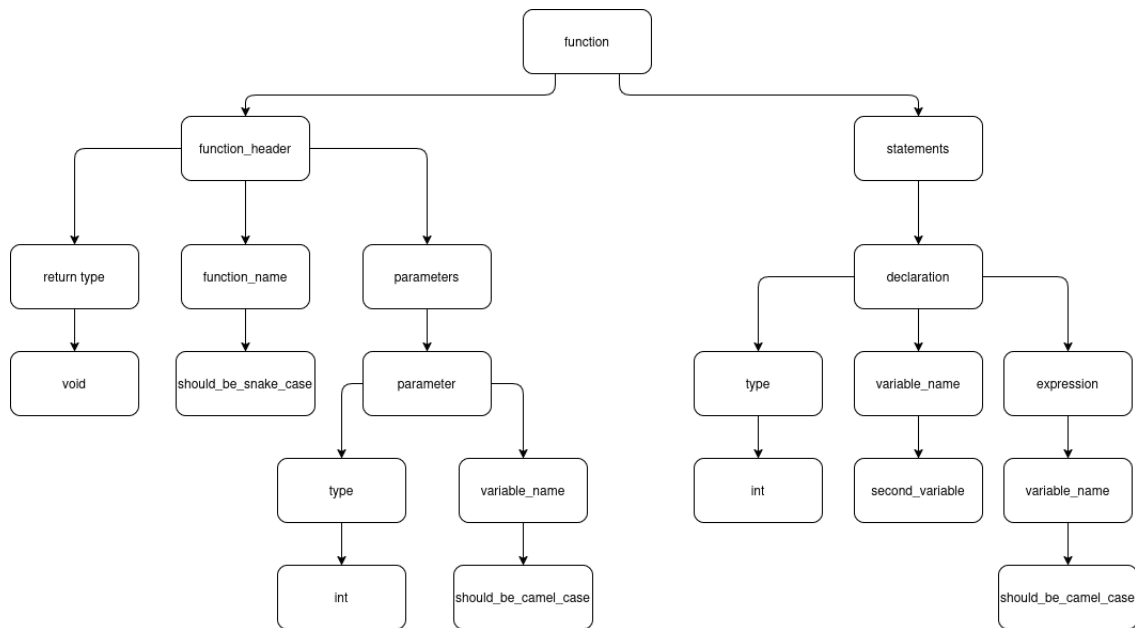


Figure 1: AST example

This AST contains a lot of information about the program structure and allows to distinguish between variable and function names. ¹

The following code should give a general idea of how transformations could be implemented on the AST. The API of `gp-modifiable-ast` is different from this example.

```

1 AST ast = parser.parse("example.java");
2 List<ASTNode> nodes = ast.find("variable_name");
3 renameToCamelCase(nodes);
4 ast.transformBackToSourceCode();

```

Listing 2: Example of transformation

This transformation would result in the following code.

¹Depending on the grammar and language, this differentiation may not always be possible on the generated AST. For example in JavaScript, functions can be passed as parameters into other functions: `function_a(function_b, variable_a)`. The implemented LR(1) parser is not able to differentiate between the types of the parameters in this case.

```
1 void should_be_snake_case(int shouldBeCamelCase) {  
2     int secondVariable = shouldBeCamelCase;  
3 }
```

Listing 3: Example of transformation

This example should illustrate why rewritable abstract syntax trees can be a useful tool for software development, because they allow a wide range of refactorings by providing access to different pieces of information about the program structure.

1.2 Related work

The library implemented for this thesis (`gp-modifiable-ast`) has similarities to common parser generators such as SableCC [sab], ANTLR [ant], GNU Bison [gnu] and others. These parser generators are mainly used for compiler development, therefore they have a different purpose than this paper. They do not need to be able to convert the AST back to the source code while maintaining the formatting and comments. Trying to use these tools for this purpose

There are custom parsers like Esprima [esp] that can generate a modifiable AST for JavaScript sources, but these are limited to a specific programming language. Roslyn [ros] is an implementation of the C# and Visual Basic compilers, which also provides an API for analysis and code manipulation. These tools are different from this thesis, as they are limited to one or a few specific programming languages. They are not intended to be used for a fast variety of languages.

This work is based on the conference paper by Jeffrey L. Overbey and Ralph E. Johnson with the title "Generating Rewritable Abstract Syntax Trees" [OJ09]. This paper defines a possible grammar definition for rewritable abstract syntax trees, which is partially implemented by `gp-modifiable-ast`. Some of the proposed definitions of [OJ09] are not implemented, as they would not make sense with the structure of `gp-modifiable-ast`. The chapter 3.5 discusses which are implemented and which are not.

2 Theoretical background

This chapter establishes the prerequisites required to understand this thesis.

2.1 Regular Expressions and Grammars

Based on the paper of Naom Chomsky [Cho56] a grammar G consists of four components.

1. A finite set N of nonterminal symbols.
2. A finite set Σ of terminal symbols.
3. A finite set P of production rules, each defined as $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, where $*$ is the kleene closure operator [HP22, page 15]².
4. A start symbol $S \in N$.

A grammar can be described as a set of definitions and rules, which construct a language L .

By limiting the production rules P , several classes can be defined based on the Chomsky hierarchy [Lin17, pages 512-513]. The main classes used in this thesis are the set of regular languages and the set of context free languages.

A language is a regular language, if it can be constructed by a regular grammar. A grammar is a regular grammar, if for all productions $p \rightarrow q$ in P the following applies: $p \in N$ and $q \in \Sigma \cup \Sigma N$ [HP22, pages 32].

A language is a context free language, if it can be constructed by a context free grammar. A grammar is context free, if for all productions $p \rightarrow q$ in P the following applies: $p \in N$ [HP22, page 37].

Regular expressions create the class of regular languages. Therefore, they can construct any language a regular grammar is able to construct [HP22, pages 25-32]. Regular expressions can be defined in a short and compact way. The following statement is a regular expression:

$$a^*ba^*b$$

This would construct all words that have any amount of "a", followed by exactly one b, followed by any amount of "a", followed by exactly one b.

However, regular languages are not sufficient to describe the syntax of a programming language. Regular languages are not able to describe the language of opening and closing brackets for example. This language requires for any opening bracket "(" a closing bracket ")". A proof, that the language $a^n b^n$ with $n \in \mathbb{N}$ is not regular can be found in [HP22, page 30].

To handle these situations context free grammars are required, they are able to parse more languages and are sufficient enough, to parse most programming languages.

An example for a context free grammar:

²The kleene operator indicates, that an element out of the set can appear zero or more times.

Let $N = \{E\}$, $\Sigma = \{a, b\}$, $S = \{E\}$ and the productions defined as:

$$P = \{E \rightarrow aEb, \\ E \rightarrow abE, \\ E \rightarrow \epsilon\}$$

ϵ is a special symbol, it refers to the empty word. That means, that E can be derived to nothing. Without this, the grammar would have endless self recursion.

This grammar would be equivalent to the opening and closing bracket example. This will parse any string, which has for one opening a exactly one closing b .

2.2 Lexer

The parsing process of a compiler is often divided into two parts. The first part is the lexer. The task of the lexer is to take the whole source code and create a token stream. The lexer works with regular expressions and avoids more complex implementations.

The lexer is a preparation for the parser, since it usually removes unwanted characters like whitespace and generalizes other tokens. For example, any string matching the regular expression $[a-zA-Z_][a-zA-Z_0-9]^*$ could be an identifier, and $[0-9]^+$ would be an integer. The name identifier is often used for variable, function and class names in lexers. This way the parser only has to work with the abstraction, not with the real input.

The following example illustrates the purpose of the lexer process

```
1 int get_fixed_sum(int a) {
2     return a + 19284;
3 }
```

Listing 4: "Example input for the lexer"

The lexer will start parsing at the beginning, identify the first token, and continue. The resulting token stream might look like this.

```
1 int_type
2 identifier
3 bracket_open
4 int_type
5 identifier
6 bracket_close
7 curly_bracket_open
8 return_stmt
9 identifier
10 add
```

```
11 number
12 semicolon
13 curly_bracket_close
```

Listing 5: "Example output of the lexer"

Each line is a token. These tokens are used as terminal symbols in the grammar definition.

The implemented lexer works very similar, the only difference is that all tokens are maintained, including whitespaces and comments.

Further informations about the functionality and implementation of a lexer can be found in [ALSU06, pages 109-189].

2.3 Parser

The parsing process is done by applying the grammar rules on the token stream. There are many types of parsers, two common ones used in compiler development are:

1. LL(1): Scanning the input from left to right, applying leftmost derivation, using one token of lookahead.
2. LR(1): Scanning the input from left to right, applying rightmost derivation, using one token of lookahead.

There are two goals of the parsing process. The first is to verify that the input conforms to the grammar. The second is to generate a tree structure that represents the input. The tree structure generated by the parser is called a Concrete Syntax Tree (CST). This tree structure is very verbose and represents the grammar definition exactly.

There are two main ways to parse source code. LL(1) parsing is an example of top-down parsing. In top-down parsing, the root node is created first, followed by the child nodes, and the leaf nodes are created at the end. LR(1) is an example of a bottom-up parsing algorithm. In this parsing strategy, the leaf nodes are generated first, and the root node is generated at the very end.

LL(1) has the advantage of being relatively easy to understand, but is less powerful than LR(1). LR(1) parsers are harder to understand and debug, but can parse more grammars than LL(1).

Each of the presented parsers are not able to parse all context free grammars, but they are sufficient enough to parse most programming languages. LR(1) is a parsing strategy that is able to parse many grammars, but requires a large amount of memory and computation to parse correctly.

Since memory and runtime are not the main priority in this thesis, a LR(1) parser was implemented to allow a large number of grammars to be parsed.

The LR(1) parser was first described by [Knu65].

The LR(1) parsing strategy relies on two different tables. The construction of the tables is explained in [ALSU06, pages 259-266].

The following example is taken from [ALSU06, pages 263, 265, 266].

Let $N = \{S, C\}$, $\Sigma = \{c, d\}$, $S = S$ be a grammar with the productions P defined as

$$P = \{S \rightarrow C C \quad (1)$$

$$C \rightarrow c C \quad (2)$$

$$C \rightarrow d\} \quad (3)$$

Let $r1$ be a reference to the production rule $S \rightarrow C C$, $r2$ be a reference to $C \rightarrow c C$ and $r3$ be a reference to $C \rightarrow d$.

The first table is called the action table.

State	c	d	$\$$
0	s3	s4	
1			acc
2	s6	s7	
3	s3	s4	
4	r3	r3	
5			r1
6	s6	s7	
7			r3
8	r2	r2	
9			r2

Table 1: LR(1) action table

The rows represent different states. The columns represent the terminal symbols of the grammar. The value of each cell can be empty, a shift, a reduce or an accept action. The $\$$ symbol is a special symbol that refers to the end of input.

A shift action will consume a token from the token stream and move to a new state. For example, s6 will perform a shift and move to state 6.

A reduce action returns to a previously encountered state and references a grammar rule that is being reduced.

State	<i>S</i>	<i>C</i>
0	1	2
1		
2		5
3		8
4		
5		
6		9
7		
8		
9		

Table 2: LR(1) goto table

An accept action is defined only once in the entire table. Once it is encountered, the parsing process is complete and the input has been successfully parsed.

The second table is the goto table.

The columns represent the non-terminal symbols of the grammar. The content of each cell can be either a reference to another state or empty.

The parser itself maintains a stack of states. The top of the stack is the current state being processed. The parser reads the current token from the token stream provided by the lexer and gets the current action from the action table. Depending on the action type, different behaviors apply.

On a shift action the parser will shift the position in the token stream and continue in the next iteration with a new token. The new state defined in the shift action is pushed to the stack. Then the next iteration is started.

On a reduce action, the parser will pop n states from the stack, where n is equal to the number of terminal and nonterminal symbols to the right of the grammar rule. Then the new top of stack is read and the state defined in the goto table for the left hand symbol of the reduced grammar rule is pushed onto the stack.

On an accept action, the parser will complete the parsing process by accepting the input.

For example, the parsing process of the word `cdcd` will be performed the steps shown in table 3.

The algorithms for the LL(1) and LR(1) parser is described in further detail in [App02, pages 45-67] and in [ALSU06, chapter 4 - Syntax analysis].

The LR(1) parsing algorithm is implemented manually, as customizations to this parsing process will be required to attach nodes to the abstract syntax tree which are not part of

Iteration	State stack	Action	Goto	Token stream	Description
1	0	s3		cdcd	Perform a shift to s3.
2	0 3	s4		dcd	Perform a shift to s4.
3	0 3 4 0 3	r3	8	cd cd	Reduce $C \rightarrow d$ Go to state 8
4	0 3 8 0	r2	2	cd cd	Reduce $C \rightarrow cC$ Go to state 2
5	0 2	s6		cd	Perform a shift to s6.
6	0 2 6	s7		d	Perform a shift to s7.
7	0 2 6 7 0 2 6	r3	9	\$ \$	Reduce $C \rightarrow d$ Go to state 9
8	0 2 6 9 0 2	r2	5	\$ \$	Reduce $C \rightarrow cC$ Go to state 5
9	0 2 5 0	r1	1	\$ \$	Reduce $S \rightarrow CC$ Go to state 1
10	0 1	acc		\$	Accept the input.

Table 3: LR(1) parse example

the grammar specification. These customizations are described in a later chapter.

The implementation is based on the algorithms described in [sta, Handouts 10, 11].

3 Implementation

The program is implemented as a Java library and can be found in this github repository [Dobb]. The library is distributed as a `jar` file. This thesis references the tag 1.0 of the library.

In this chapter, a `user` is a software developer who uses this library.

3.1 Architecture

The only dependencies used are JUnit and JUnit-jupiter [jun] for testing purposes.

The following figure illustrates the architecture and design of the `gp-modifiable-ast` library. Each small rectangle represents an entity of the entire program, the small rectangles with **bold** text are user-defined inputs and implementations and the ones with *italic* text are implementations of the `gp-modifiable-ast` library.

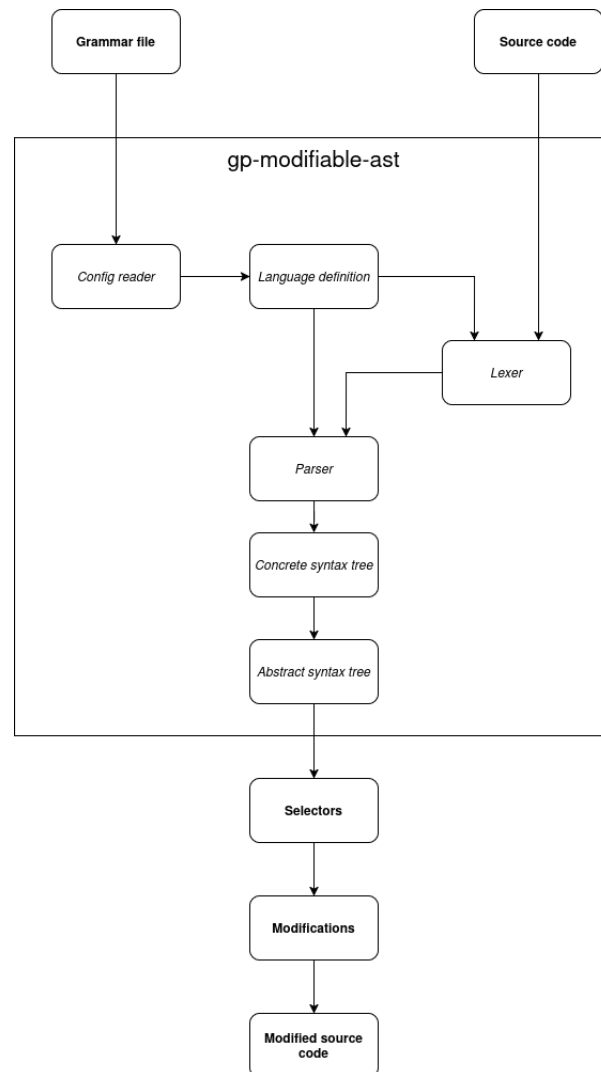


Figure 2: Architecture of `gp-modifiable-ast`

The user defines a grammar file for a given programming language and provides the source code to `gp-modifiable-ast`.

`gp-modifiable-ast` first parses the grammar file and creates a language definition that contains the grammar productions, lexer rules, and generic settings for the language.

This language definition is then passed along with the source code by the user to the lexer process, which passes the token stream and the language definition to the parser process. The parser process parses the token stream, applies the grammar rules and creates the concrete syntax tree.

In the next step, the concrete syntax tree and the grammar definition will be used to create

the abstract syntax tree.

This is the output of the `gp-modifiable-ast` library. By using selectors, which will be defined later, the user can perform search actions on the AST to find specific nodes. On the nodes found, the user can apply modifications and convert the AST back to source code.

The following sections will describe the behaviour and architecture of the components.

3.2 Class generation vs general classes

Most parser generators will generate source code for the parser. This allows the user to modify the generated classes, for example by adding new methods. This allows better usability if done correctly, and better performance. Instead of generating code for the parser, `gp-modifiable-ast` will use generic classes that are the same for all grammars being parsed. This approach does not require any additional code generation by the parser and allows an easier introduction to the system, since a language definition and a source file can simply be passed to the parser generator and an AST is generated. Otherwise, the parser must first be generated and then the parser must be called to parse the source file.

3.3 Grammar file syntax

The productions and lexer definitions are defined separately from the implementations in their own text files. Each language has exactly one text file defining the productions and lexer definitions. The contents are divided into four different chapters.

The following example shows a grammar that could be a part of a programming language grammar, which allows to parse simple arithmetic expressions and allows comments and whitespaces.

```
1 LANGUAGE_DEF
2     grammar_start = S;
3
4 LEXER_RULES
5     add = "+";
6     subtract = "-";
7     multiply = "*";
8     divide = "/";
9     bracket_open = "(";
10    bracket_close = ")";
11    integer = regex(\d+);
12    whitespace = regex(\s+);
13    single_line_comment = customMatcher(singleLineCommentMatcher);
14    multi_line_comment = customMatcher(multiLineCommentMatcher);
```

```

15
16 HIDDEN_LEXER_RULES
17     whitespace, single_line_comment, multi_line_comment;
18
19 PRODUCTIONS
20     S    ->  S[alias=left] add T[alias=right] |
21             S[alias=left] subtract T[alias=right] |
22             T;
23     T    ->  T[alias=left] multiply F[alias=right] |
24             T[alias=left] divide F[alias=right] |
25             F;
26     F    ->  bracket_open S bracket_close |
27             integer;

```

Listing 6: Grammar file example

This grammar file would be able to parse the following sample code:

```

1 // Arithmetic grammar test
2 (5+10) /** inline comment */
3 * (15*20)

```

Listing 7: Grammar file example

The first chapter is `LANGUAGE_DEF`. This contains some general definitions for the language. This chapter contains e.g. how comment styles are identified, the name of the language, whether the language is case-sensitive, which production should be used as the starting production, and whether the language is case-sensitive or not. All settings here are defined as key-value pairs. The available options are `name`, `single_line_comment_style`, `multi_line_comment_style_start`, `multi_line_comment_style_end`, `case_sensitive`, `grammar_start`.

The second chapter is the `LEXER_RULES`. This defines all the tokens that can occur in the source file and that the lexer should handle. The lexer can handle three different types of definitions. The first type matches a fixed string. The second type matches a regular expression. The last type uses a custom implementation to check for a match. The last matching definition wins. For example, the token definition `true_literal = "true"`; and the definition `identifier = regex([a-zA-Z_][a-zA-Z_0-9]*)`; would both match the string `true`. Therefore, the more general `identifier` token should be defined before the `true_literal` token. This will cause the lexer to recognize the text `true` as `true_literal`. A lexer key must be written in all lowercase letters and underscores.

The chapter `HIDDEN_LEXER_RULES` is a comma separated list of lexer definition keys, that should not be handled in the parsing process and should not be visible in the AST by default. This category should be used mainly for whitespace and comment tokens. If these are not added to the category, the grammar rules must handle them. Since whitespace and comments can be added basically anywhere, this would result in a very verbose definition of productions.

The chapter `GRAMMAR_RULES` contains all productions of this language. The syntax is similar, but not identical, to the backus naur form [ALSU06]. The syntax has been extended with most of the modifiers suggested by [OJ09]. The terminals are the keys of the lexer definitions. The nonterminals are defined by the left sides of the productions. All nonterminals must be written in uppercase and underscore. `EPSILON` is used for the empty word. Each terminal and nonterminal symbol can be extended by a list of modifiers in square brackets after the name.

`S -> S[alias=left] add T[alias=right]` defines the production with `S` on the left and `S add T` on the right. `S` refers to the same production, `T` refers to a different production. `add` refers to a lexer definition. `[alias=right]` sets an alias in the AST for the `T` production.

3.4 CST-Generation

The concrete syntax tree (CST) is a tree structure that exactly matches the grammar and lexer definitions. The generated CST should also contain nodes that are usually omitted, such as comments or whitespaces.

The CST is implemented by a `ConcreteSyntaxTreeNode` class. A node can refer to either a production or a lexer token. We keep the productions because they allow us to make advanced modifications and generate a less verbose abstract syntax tree from the CST. We keep the lexer tokens because they contain the original sources.

This structure is generated directly by the parser. The parser maintains a second stack containing tree nodes that are not bound to a parent node.

As the parser passes through the token stream provided by the lexer, it generates new tree nodes and pushes them onto the stack whenever either the token is in `HIDDEN_LEXER_RULES` or the action performed is a shift action.

On a reduce action the parser will also create a new tree node. This tree node will have other tree nodes from the top of the stack as children. This behavior is very similar to how the parser manages its state. The main difference is the tokens in the `HIDDEN_LEXER_RULES`. For example, the reduced production may have only one symbol on the right side, but on the tree node stack there are two nodes from `HIDDEN_LEXER_RULES`. In this case the parser will pop three states from the top of the tree node stack and add them as children of the newly created node in reverse order. Another difference is that when the parser reaches the `acc` state, it is not guaranteed that the tree node stack contains only the root node. There may be tokens from `HIDDEN_LEXER_RULES` which appeared at the beginning of the parsed string. So the parser will find the root node and prepend all other entries in the tree node stack to the children of the root node.

With this algorithm, every single token from the lexer's token list is transformed into a tree

node and placed in the CST. This guarantees that the whole source code is represented in the CST and that the CST could be transformed back to the source code exactly.

The following code shows a simplified version of the CST generation during the parser process.

```

1  if(isHiddenLexerRule() || isShiftAction()) {
2      danglingTreeNodeStack.push(new TreeNode(currentParseAction));
3  }
4
5  if(isReduceAction()) {
6      TreeNode newTopOfStack = new TreeNode(currentParseAction);
7      int i = getRightHandSymbolCount();
8      while(i > 0) {
9          TreeNode currentTopOfStack = danglingTreeNodeStack.pop();
10         newTopOfStack.pushChild(currentTopOfStack);
11         if(!currentTopOfStack.isHiddenLexerRule()) {
12             i--;
13         }
14     }
15     danglingTreeNodeStack.push(newTopOfStack);
16 }
17
18 if(isAcceptAction()) {
19     TreeNode rootNode = extractRootNode(danglingTreeNodeStack);
20
21     while(!danglingTreeNodeStack.isEmpty()) {
22         TreeNode treeNode = danglingTreeNodeStack.pop();
23         rootNode.prependChild(treeNode);
24     }
25
26     return rootNode;
27 }

```

Listing 8: CST Generation

3.5 AST-Generation

Even though the CST already contains all relevant information about the sources, it is very verbose, and the structure of the CST is usually not very friendly to work with. For this reason, `gp-modifiable-ast` implements the modifiers provided by [OJ09], which are used to convert the CST into an AST.

There are three types of nodes in the AST. `ProductionTreeNode` are nodes derived from a grammar production. `TokenTreeNode` are nodes derived from a lexer token. `StringTreeNode` are nodes that contain a string and are used to replace other tree nodes. The `StringTreeNode` is not part of the initial AST. This node can be used by the user to replace subtrees in the final AST or to add a new node.

Any `TokenTreeNode` that references a token from `HIDDEN_LEXER_RULES` is marked as hidden. These nodes are stored in the AST, but are not visible to the user unless specifically requested. Next, the AST structure is modified by the modifiers defined in the grammar file. Each modifier has its own implementation and is applied bottom-up for each node.

3.5.1 list-Modifier

The `list` modifier, proposed by [OJ09], is a modifier designed to flatten out self-recursive productions. This modifier can only be applied to symbols on the left side of a production. The following example could be a production for the function parameters in common programming languages.

```
1 PARAMETER_LIST[list] -> PARAMETER_LIST PARAMETER |
2                           PARAMETER |
3                           EPSILON;
```

Listing 9: list modifier example

This grammar rule generates the following CST for an input with for example three parameters.

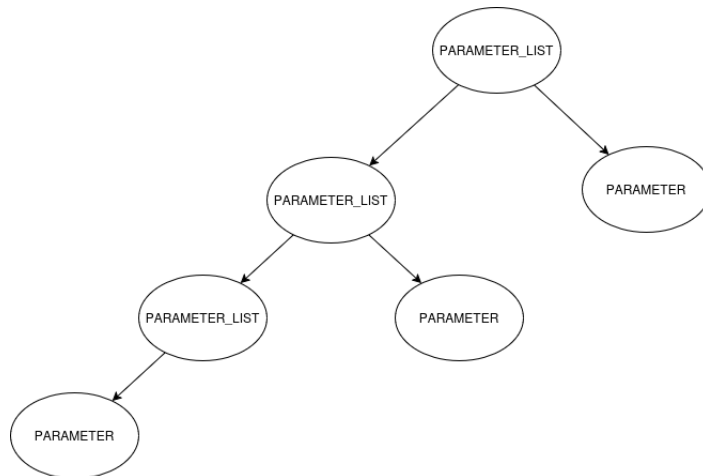


Figure 3: CST before the list modifier is applied

This structure is too verbose and difficult to work with in practice, so by using the `[list]` modifier, the following AST gets generated.

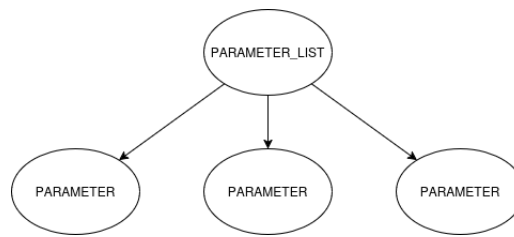


Figure 4: AST after the list modifier is applied

This structure is much easier to manage and still contains all relevant information about the sources.

The modifier is applied by checking if the children of the current `ProductionTreeNode` n contain a `ProductionTreeNode` m that references the same production. If so, m is replaced in the children list of n by all children of m . Since the modifiers are applied bottom-up, this works for nested trees.

3.5.2 alias-Modifier

The `alias` modifier, proposed by [OJ09], is a modifier that can be applied to any symbol on the right hand side of a production. This modifier, adds an alias to the tree node that can be used to search the tree. The following could be an example of a production for addition in a programming language.

```
ADD -> NUMBER[alias=left] plus NUMBER[alias=right];
```

Without the `alias` modifier, the only way to distinguish between the two `NUMBER` nodes would be by the order of the children of the `ADD` tree node. The `alias` modifier allows for cleaner searches in the AST.

This modifier is applied by simply storing the alias in the tree node.

3.5.3 hidden-Modifier

The `hidden` modifier, proposed by [OJ09], is a modifier that can be applied to terminal symbols on the right hand side of a production. This modifier will hide the tree node in the AST. In the previous example of the `ADD` production, the `ADD` tree node would have a `TokenTreeNode` child that references the `plus` lexer definition. This information is obsolete because the production will always contain this node, and the production name already

contains the necessary information. By applying the `hidden` modifier to the `plus` symbol, the corresponding tree node will still exist, but will not be visible unless specifically requested.

3.5.4 inline-Modifier

The `inline` modifier, proposed by [OJ09], is a modifier that can be applied to nonterminal symbols on the right hand side of a production.

Applying the modifier will replace the node with all its children. This modifier should be used on nonterminals that do not carry important information themselves, but their children do. This way all information is preserved, but the tree structure is simplified.

The following grammar should illustrate the purpose of this modifier.

```

1 EXPRESSION          ->    ARITHMETIC_EXPRESSION[inline];
2 ARITHMETIC_EXPRESSION ->    PLUS_EXPRESSION;
3 PLUS_EXPRESSION      ->    integer plus integer;
```

Listing 10: inline modifier example grammar

The `ARITHMETIC_EXPRESSION` production is only used for creating the grammar in this case, as it may not be possible to create an LR(1) grammar without it. Another reason for defining a grammar this way is to prevent duplications. If `ARITHMETIC_EXPRESSION` is also used at other places in the grammar file, the extraction into a new production can reduce the total size of a grammar definition and improve the maintainability.

This production carries however no information itself that are relevant to the program structure, therefore it can be omitted in the AST.

This modifier is implemented by replacing all `ProductionTreeNodes` that have this modifier with their corresponding children.

Therefore, the resulting AST does not contain the `ARITHMETIC_EXPRESSION` node.

3.5.5 Not implemented modifiers

[OJ09] proposed additional modifiers that are not implemented in `gp-modifiable-ast`. The first is the `Boolean Access` modifier, which would replace a node with a boolean value depending on whether that symbol was parsed. However, since `gp-modifiable-ast` do not generate classes for every production, this would be of little use. The same reason applies to the `superclass` modifier, which would create a hierarchy in the generated classes of the parser generator.

3.6 AST search methods

Before any changes can be made to the AST, the nodes that need to be changed must first be found.

For this approach, each AST node has three methods.

1. `query: Selector -> QueryResult`
2. `queryChildren: Selector -> QueryResult`
3. `queryImmediateChildren: Selector -> QueryResult`

The `query` method takes a selector and returns a `QueryResult`. The `QueryResult` contains all nodes in the subtree of the searched node that match the selector. The `QueryResult` may also contain the searched node itself if it matches the selector.

The `queryChildren` behaves the same way, but does not include the searched node itself.

The `queryImmediateChildren` will return only the immediate children of the searched node that match the selector.

These methods are able to find hidden nodes.

The `QueryResult` instance also allows to perform queries on the result. This will perform the corresponding method on all nodes in the result and create a new `QueryResult` instance containing the merged results of each node. This allows for easy chaining of selectors. You can also merge `QueryResult` instances for further processing with the `merge` function.

3.7 Selectors

A `Selector` is a class instance that should test whether a given tree node matches the rules defined by the `Selector`. The `Selectors` can be divided into three different categories.

The first category are logical `Selectors`. They should represent logical operations. There are two `Selectors` defined by default, `AndSelector` and `OrSelector`. Each takes a list of other selectors. The `AndSelector` will only match tree nodes that match all passed `Selectors`. The `OrSelector` will match all tree nodes where at least one of the passed `Selectors` matches the node.

The second category are data `Selectors`. These selectors search for nodes based on the data they contain. They can match the name of a production, the alias of a production, the

name of a lexer definition, and the value of a lexer token. The implemented selectors are `AliasSelector`, `ProductionSelector`, `TokenSelector`, `TokenValueSelector`. All of these take a string as a parameter and check the corresponding notes for that string.

The last category are structural `Selectors`. These selectors search for nodes that have specific nodes in their children or parents.

The following example of an AST is intended to illustrate the purpose of this category.

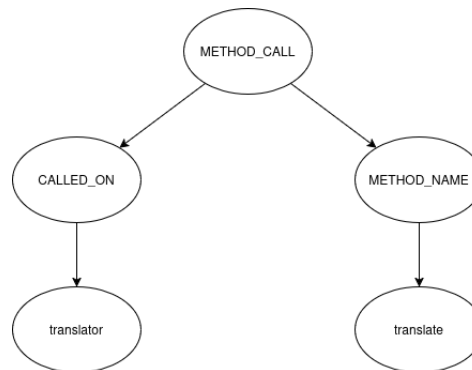


Figure 5: AST example for selectors

To search for all `METHOD_CALL` nodes, that call `translate` on a `translator` object, these types of selectors will be used.

An example selector for this would be

```

1 new AndSelector(
2   new ProductionSelector("METHOD_CALL"),
3   new HasImmediateChildSelector(
4     new AndSelector(
5       new ProductionSelector("CALLED_ON"),
6       new HasImmediateChildSelector(
7         new TokenValueSelector("translator")
8       )
9     )
10  ),
11  new HasImmediateChildSelector(
12    new AndSelector(
13      new ProductionSelector("METHOD_NAME"),
14      new HasImmediateChildSelector(
15        new TokenValueSelector("translate")
16      )
17    )
18  )
19 )
  
```

Listing 11: Selector example

By calling the `query` method on the root node of the AST, all the `METHOD_CALL` nodes in the AST that match those conditions will be returned.

3.8 AST Modifications

The `AbstractSyntaxTreeNode` class contains several methods for making changes.

1. `replace` Replaces this node with one node or a list of other nodes.
2. `replaceChild` Replaces a node in its children with one node or a list of other nodes.
3. `remove` Removes this node and its subtree.
4. `removeChild` Removes a child.
5. `addChild` Adds a child to the end of the child list.
6. `deepClone` Clones this node and all its children, returning a node that is the same as the current one, but without a parent reference.

Each `AbstractSyntaxTreeNode` references its corresponding parent node. To maintain integrity, a user can only add nodes that do not have a parent node, and can only replace nodes with a parent reference. To add a node to multiple other nodes, a user can clone the node to be added.

3.9 Code generation

After adjustments have been made, the AST can be converted back to source code.

This process is done by performing a depth first search on the AST. The parsing and AST generation process ensures that all information from the original source file is transferred to the AST. The leaves of the AST are either `StringTreeNode` instances or `TokenTreeNode` instances.

If a node is a `StringTreeNode` instance, the user has defined the string it represents.

If a node is a `TokenTreeNode` instance, the lexer token is stored inside the node. The lexer token contains the source code that constructed that token.

Therefore, the code can be generated by concatenating the strings that represent the tree nodes. The code can be obtained by calling `getSourceCode` on an AST node.

3.10 Extendability

To allow for possible extensibility, gp-modifiable-ast allow for several types of customization

3.10.1 Lexer

As mentioned before, lexer definitions can contain custom matchers. These matchers are custom implementations that are referenced by a name in the grammar file. These classes are written by the user and registered by calling `CustomMatcherRegistry.registerCustomMatcher`. A custom matcher must implement the `ICustomMatcher` interface. A matcher gets a `LexerContext` and returns `null` or the matched string. Using the `LexerContext` parameter, the matcher can receive the remaining input and check if the beginning of the remaining input can be tokenized by this matcher.

The following example is an implementation of a custom matcher, that would match single line comments.

```
1 public class ExampleCustomMatcher implements ICustomMatcher {
2     @Override
3     public String match(LexerContext context) {
4         if(context.getNextNChars(2).equals("//")) {
5             // Find the end of line. This will be the comment token
6             String remaining = context.getRemainingSource();
7             Pattern p = Pattern.compile(".*(\\r\\n|\\r|\\n|$)");
8             Matcher matcher = p.matcher(remaining);
9             if (matcher.find()) {
10                 return matcher.group();
11             }
12         }
13         return null;
14     }
15 }
16 }
```

Listing 12: Example of a custom matcher

3.10.2 AST Generation

We allow several ways to customize AST generation. First, a user can implement custom modifiers for grammar symbols. You can also post-process and decorate the AST. For example, a user can replace the tree nodes with your own implementation by extending `AbstractSyntaxTreeNode`.

The following example is an implementation of a tree node that is simply a string. This tree node can be used to add or replace code in the AST.

```
1 public class StringTreeNode extends AbstractSyntaxTreeNode {
2     private final String value;
3
4     public StringTreeNode(String value) {
5         this.value = value;
6     }
7
8     public String getDisplayValue() {
9         return "StringTreeNode, value: " + this.value;
10    }
11
12    public String getValue() {
13        return value;
14    }
15
16    protected String getSources() {
17        return this.value;
18    }
19 }
```

Listing 13: Implementation of StringTreeNode

3.10.3 Selectors

Custom selectors can be defined by extending BaseSelector. These can be easily defined and used with the query methods of the tree node.

The following example shows the implementation of a selector that matches a tree node based on the production used to create that node.

```
1 public class ProductionSelector extends BaseSelector {
2     private final String production;
3
4     public ProductionSelector(String production) {
5         this.production = production;
6     }
7
8     @Override
9     public boolean matches(AbstractSyntaxTreeNode treeNode) {
10        if (treeNode instanceof ProductionTreeNode convertedNode) {
11            return convertedNode.getProduction()
12                .leftHandSymbol()
13                .name()
14                .equals(this.production);
15        }
16        return false;
17    }
18 }
```

Listing 14: Custom selector implementation

4 Implemented Grammars

There are two grammars implemented in `gp-modifiable-ast` by default. The first grammar is MiniJava. The MiniJava language was defined by [App02].

The second grammar is an extension to MiniJava, which was defined for this thesis. This extended grammar allows for more operations in comparisons. This grammar is only used to illustrate some possible refactorings and use cases of `gp-modifiable-ast`.

4.1 MiniJava

The implemented grammar is based on the transformed grammar in backus naur form from [min].

The MiniJava grammar has been defined by `gp-modifiable-ast`. This is a very limited subset of Java, but it represents common constructs found in modern programming languages. It includes basic object oriented programming, simple conditional statements, while loops and basic arithmetic operations. However, the grammar is quite limited. For example, it does not allow any other comparison operator then `<` and not more then one arithmetic operation without brackets. As these limitations make it hard, to create meaningful refactorings, the syntax of MiniJava was extended for this thesis to allow for other operations.

4.2 Extended MiniJava

The extended MiniJava grammar includes a small change to the grammar.

The grammar allows the use of more comparison operators. MiniJava only allows `<`.

Extended MiniJava includes the other comparison operators `>`, `>=`, `<=` and `==`.

Each of those comparison operators can be refactored to `<`, as MiniJava only supports integers and not floating point values.

The purpose of extended MiniJava is to perform a source code transformation, which converts any source code written in extended MiniJava to the the MiniJava syntax. These types of transformations are commonly used in practice, for example by Babel [bab], which is used to transform modern JavaScript into old JavaScript syntax. This allows developers to write code using the latest JavaScript standard while still supporting old browsers, that run the transformed code.

5 Use cases

This chapter contains some refactorings that have been implemented on MiniJava and some refactoring examples are shown, which have not been implemented due to the limited grammar of MiniJava.

The implementation of the examples can be found here [Dobc].

5.1 Yoda conditions

Yoda conditions are conditions in the form `a compop b`, where `a` is a literal (e.g. 5) and `b` is a expression [DB11, page 71]. Non Yoda conditions are conditions in the form `b compop a`, where `a` is a literal and `b` is a expression.

Usually in software development, you want to follow one style and use it for the whole project. Therefore, various IDEs and other tools like lexers exist, which check the styling and if possible fix them automatically. These tools may not exist for domain specific languages, therefore a usecase of this project can be those refactorings.

To implement this refactoring, the extended MiniJava syntax is used. First, a selector is applied to find all comparison operators in the AST. From the result, the left and right sides of the comparison node can be obtained by an alias selector. If the left side is an integer literal, the left node is replaced by the right node and the right node is replaced by the left node. Finally, the `compop` must be adjusted to reflect the changes. `<` will become `>`, `>` will become `<`, `<=` will become `>=`, `>=` will become `<=`. `==` will not need to be adjusted.

Since whitespaces are real nodes in the AST, this refactoring does not mess with formatting. All whitespace within the are kept exactly where they were in the original sources. The node corresponding to `a` contains only the `a` constant. Any whitespace between `a` and `compop` is a separate node in the AST. Since `b` can be any expression, `b` can also contain whitespace. For example, `b` could be `"(a + 5)"`. These whitespaces are children of the `b` expression, and therefore they keep their position relative to the `b` node. Therefore, swapping `a` and `b` does not change the formatting of the file.

The following code is a snippet from the actual refactoring implemented in [Dobd]

```

1 if(leftIsLiteral && !rightIsLiteral) {
2     left.replace(right.deepClone());
3     right.replace(left.deepClone());
4
5     String newCompOp = compop.getValue();
6     switch (compop.getValue()) {
7         case "<":
8             newCompOp = ">";
9             break;
10        case "<=":

```

```
11         newCompOp = ">=";  
12         break;  
13     case ">":  
14         newCompOp = "<";  
15         break;  
16     case ">=":  
17         newCompOp = "<=";  
18         break;  
19     }  
20     compop.replace(new StringTreeNode(newCompOp));  
21 }
```

Listing 15: Refactor Yoda conditions

5.2 Translation system

This is an example for a refactoring, which may be required to be performed on a large scale and that is not doable by common tools.

Let a translation system be a system, which stores some data for some keys. This data includes the translations for each desired language and parameters. A parameter is a variable, which gets dynamically inserted into the text. One of the parameters can be used as a pluralization parameter.

For example, the key "apple" resolves to the english translation "{NAME} has {COUNT} apple" if the parameter COUNT is 1 and "{NAME} has {COUNT} apples" if COUNT is not 1. {NAME} is a second parameter, which does not require any pluralization.

The wanted translation is received by calling `translate("apple", {NAME: user, COUNT: i})`.

When switching to another translation system, which requires the pluralization parameter to be passed separately, an issue occurs. The desired call syntax will be: `translate("apple", {NAME: user}, {COUNT: i})`

The old code does not include any information, which of the parameters is the pluralization parameter. We require the connection to an external source to correctly identify the wanted parameter.

This refactoring might be possible with regular expressions, but it would require a separate regular expression for each translation token. Even if such a regular expressions were generated for each translation token, no regular expression will be able to parse all possible call syntaxes in most programming languages. Therefore, not all replacements can be done by regular expressions, and only part of the refactoring will be done. In the worst case, a regular expression might match unrelated code and introduce new problems.

The refactoring could be done manually, but this will be very time consuming if there

are many translations. Another rather simple solution would be to implement the adapter pattern [Sar16, pages 47-52]. This design pattern allows us to use the old call syntax and the adapter translates it to the new call syntax at runtime. However, this solution would leave us with outdated code. The final option is to implement a custom solution.

With `gp-modifiable-ast`, the sources can be parsed given a grammar, receive an AST, and identify the translation key and any parameters passed. To identify which parameter is the pluralization parameter, the definition of the translation can be received. Then the pluralization parameter can be removed from the call syntax and added as a new parameter to the function call.

This refactoring has not yet been implemented, as both the MiniJava and extended MiniJava grammars are not sufficient to support an illustrative example.

5.3 Code transpilation

This project may also be used to transpile code from a source grammar to a target grammar. As an example, a transpiler was written, which transpiles applications that use the extended MiniJava grammar to applications that use the regular MiniJava grammar. An example of a transpiler which performs operations like these would be `babel` [bab]. This tool is used to transpile a newer version of JavaScript into an older one, which can be executed by more browsers. That allows the developer to utilize new functionalities of the programming language while not breaking backwards compability.

The goal of this implementation is to show the power of a rewritable abstract syntax tree.

As MiniJava only supports the less then operator, every conditional that uses a different operator needs to be transformed. For this the following equivalent statements for $a, b \in \mathbb{Z}$ are used:

$$a > b \iff b < a \tag{4}$$

$$a \leq b \iff a < b + 1 \tag{5}$$

$$a \geq b \iff b \leq a \iff b < a + 1 \tag{6}$$

$$a = b \iff (a \leq b) \wedge (a \geq b) \iff (a < b + 1) \wedge (b < a + 1) \tag{7}$$

As all new comparison operators have been reduced to the less then operator, the refactoring can be implemented.

This is done by creating a selector to find all comparisons, reading the comparison operator and performing the refactoring. The following code is an implementation of the transpilation process. The complete source is accessible at [Doba].

```

1 AbstractSyntaxTreeNode ast = gpModifiableAST.createAst(input);
2 QueryResult comparisons = getComparisons();
3 for(AbstractSyntaxTreeNode comparison: comparisons) {
4     AbstractSyntaxTreeNode left = getLeftNode();
5     TokenTreeNode compop = getCompOpNode();
6     AbstractSyntaxTreeNode right = getRightNode();
7
8     switch (compop.getValue()) {
9         case "<=":
10             compop.replace(new StringTreeNode("<"));
11             right.replace(List.of(
12                 new StringTreeNode("("),
13                 // As right is still attached to the tree, we need to
14                 // clone the node.
15                 right.deepClone(),
16                 new StringTreeNode(" + 1")
17             ));
18             break;
19         case ">":
20             compop.replace(new StringTreeNode("<"));
21             left.replace(right.deepClone());
22             right.replace(left.deepClone());
23             break;
24         case ">=":
25             compop.replace(new StringTreeNode("<"));
26             left.replace(right.deepClone());
27             right.replace(List.of(
28                 new StringTreeNode("("),
29                 left.deepClone(),
30                 new StringTreeNode(" + 1")
31             ));
32             break;
33         case "==" :
34             // Replace equality operator
35             break;
36     }
37 }

```

Listing 16: Code transpilation

5.4 Code analysis

The `gp-modifiable-ast` can be used to perform code analysis. Even though this was not the main intention, the requirements of the project is equal to the ones required for code analysis.

All comments and whitespaces are kept. Specific comments are often used to disable code analysis for certain parts of code. Whitespaces are required to check code formatting.

However as the `gp-modifiable-ast` is not implemented for a specific programming language, some features cannot be implemented in general. For example a symbol table cannot be implemented for all programming languages in the same way.

The main benefit of using `gp-modifiable-ast` for code analysis would be that some of the analysis actions could be defined once and be used for multiple programming languages.

It would still be required to pass configurations for each programming language into the analysis action.

6 Conclusion and further work

The implemented library is capable of generating an AST that preserves all information about the original sources. This is done by parsing a language definition file, building a LR(1) parser, parsing the source code, and building an AST which still contains all the tokens that are usually ignored by parsers. With the provided API, it is possible to traverse and modify the AST in a way to implement custom modifications to a given source.

The grammar for MiniJava has been defined and several refactorings have been implemented to show the functionality and usability of the implementation.

A limiting factor is the effort required to define a language grammar. This could be improved by extending the grammar's syntax to accept the extended backus naur form.

Another problem, due to the fact that the library is not implemented for a specific programming language, is the lack of functionality of complete compiler frontends, including but not limited to symbol tables. If there is a way to create symbol tables and detect usages by extending the language definition file, this library could be used for more complex refactorings. Currently, these functions must be built on top of the library.

Therefore, this library is currently best used for refactorings to single subtrees of an AST that contain all the information needed to perform the refactoring.

List of Figures

1	AST example	2
2	Architecture of gp-modifiable-ast	10
3	CST before the list modifier is applied	15
4	AST after the list modifier is applied	16
5	AST example for selectors	19

List of Tables

1	LR(1) action table	7
2	LR(1) goto table	8
3	LR(1) parse example	9

References

- [ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [ant] Antlr. <https://wwwantlr.org/>. Accessed: 2024-12-30.
- [App02] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.
- [bab] Babel. <https://babeljs.io/>. Accessed: 2024-12-29.
- [Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.
- [DB11] T. Foucher D. Boswell. *The Art of Readable Code*. O'Reilly Media, Inc., November 2011.
- [Doba] Michael Doberstein. Extended minijava transformation. <https://github.com/midob101/gp-modifiable-ast-examples/blob/main/src/main/java/org/example/TransformToMiniJava.java>. Accessed: 2024-12-29.
- [Dobb] Michael Doberstein. General purpose modifiable ast parser. <https://github.com/midob101/gp-modifiable-ast>. Accessed: 2024-12-11.
- [Dobc] Michael Doberstein. gp-modifiable-ast examples. <https://github.com/midob101/gp-modifiable-ast-examples>. Accessed: 2024-12-29.
- [Dobd] Michael Doberstein. Yoda refactoring. <https://github.com/midob101/gp-modifiable-ast-examples/blob/main/src/main/java/org/example/RefactorYoda.java>. Accessed: 2024-12-29.
- [esp] esprima. <https://esprima.org/>. Accessed: 2024-12-18.
- [gnu] gnu-bison. <https://www.gnu.org/software/bison/>. Accessed: 2024-12-18.
- [HP22] W. Homeda and MW. Pedrycz. *Automata Theory and Formal Languages*. De Gruyter, 2022.
- [jun] Junit 5. <https://junit.org/junit5/>. Accessed: 2024-12-18.
- [Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
- [Lin17] Peter Linz. *An Introduction to Formal Languages and Automata, sixth edition*. JONES & BARTLETT, 2017.

- [min] Minijava bnf. <https://www.cse.iitm.ac.in/~krishna/courses/2014/even-cs6848/minijava.html>. Accessed: 2024-06-08.
- [OJ09] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, pages 114–133, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [ros] Roslyn. <https://github.com/dotnet/roslyn>. Accessed: 2024-12-30.
- [sab] sablecc. <https://sablecc.org/>. Accessed: 2024-12-18.
- [Sar16] V. Sarcar. *Java Design Patterns*. Apress, 2016.
- [sta] Stanford cs143 compilers. <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/>. Accessed: 2024-12-30.